020 Simple Functional Data Structures; ADTs in Scala

This set assumes that you are comfortable with last week's material, and you have read the book at least until section 3.4 inclusive (we shall work with Section 3.5 next week). Exercises marked **[–]** are meant to be very easy, and can be skipped by students that already know functional programming and feel comfortable. Exercises marked **[+]** are difficult (advanced programming level).

Expect to spend at least 6 hours solving these exercises, excluding the time for reading the book, making coffee, chit-chat and smoking :) The set is quite large, but many of the problems are easy, small and very nice. In fact, this week shows you the core of functional programming, the foundation for the rest of the course.

As usual, do not use variables, side effects, exceptions or return statements.

Hand-in: File `Exercises.scala` with solved Exercises (as many as you managed). Make sure that the file compiles. We don't accept Word files, PDF files, or zipped files. Make sure that you have filled in your ITU account names on top of the file.

Exercise 1 [–]. What will be the result of the following match expression?[1] Answer the question without running the code on your computer.

```
List(1,2,3,4,5) match {
  case Cons(x, Cons(2, Cons(4, _))) => x
  case Nil => 42
  case Cons(x, Cons(y, Cons(3, Cons(4, _)))) => x + y
  case Cons(h, t) => h + sum(t)
  case _ => 101
}
```

From this point, the exercise proceeds in the file `Exercises.scala`. The exercise numbers are marked in the Scala file. Solve the exercises in the order of numbers.

Exercise 2 [–]. Implement the function `tail` for removing the first element of a list. Note that the function takes constant time. What are different choices you could make in your implementation if the list is `Nil`?[2]

```
def tail[A] (as: List[A]) : List[A]
```

Use the REPL to test whether your solution works.

Exercise 3 [–]. Generalize `tail` to the function `drop`, which removes the first `n` elements from a list. Note that this function takes time proportional only to the number of elements being dropped—we do not need to make a copy of the entire list.[3]

```
def drop[A](l: List[A], n: Int): List[A]
```

Test your function in the REPL.

Exercise 4 [–]. Implement `dropWhile`, which removes elements from the given list prefix as long as they match a predicate `f`.[4]

[1] Exercise 3.1 [Chiusano, Bjarnason 2014]
[2] Exercise 3.2 [Chiusano, Bjarnason 2014]
[3] Exercise 3.4 [Chiusano, Bjarnason 2014]
[4] Exercise 3.5 [Chiusano, Bjarnason 2014]

```
def dropWhile[A](l: List[A], f: A =>Boolean): List[A]
```

Exercise 5 [–]. Implement a function, `init`, that returns a list consisting of all but the last element of the original list. So, given `List(1,2,3,4)`, `init` will return `List(1,2,3)`.

```
def init[A](l: List[A]): List[A]
```

Test the function in the REPL. Is this function constant time, like `tail`? Is it constant space?[5]

Exercise 6 [–]. Compute the length of a list using `foldRight`:[6]

```
def length[A](as: List[A]): Int
```

Exercise 7 [+]. The function `foldRight` presented in the book is not tail-recursive and will result in a `StackOverflowError` for large lists. Convince yourself that this is the case, and then write another general list-recursion function, `foldLeft`, that *is* tail-recursive:

```
def foldLeft[A,B](as: List[A], z: B)(f: (B, A) =>B) : B
```

For comparison consider that:

`foldLeft (List(1,2,3,4),0) (_ + _)` computes $(((0+1)+2)+3)+4$ while
`foldRight (List(1,2,3,4),0) (_ + _)` computes $1+(2+(3+(4+0)))$.

In this case the result is obviously the same, but not always so. The two functions also have different space usage.[7]

Exercise 8 [–]. Write `product` (computing a product of a list of integers) and a function to compute the `length` of a list using `foldLeft`. Test both functions briefly in the REPL.[8]

Exercise 9. Write a function that returns the reverse of a list (given `List(1,2,3)`, it returns `List(3,2,1)`). See if you can write it using a fold.[9]

Exercise 10 [+]. Write `foldRight` using `foldLeft` (**Hint:** reverse). This version of `foldRight` is useful because it is tail-recursive, which means it works even for large lists without overflowing the stack.

Write `foldLeft` in terms of `foldRight`. Do not use `reverse` here (reverse is a special case of `foldLeft` so a solution based on `reverse` is less interesting). **Hint:** to do this you will need to synthesize a function that computes the run of `foldLeft`, and then invoke this function. To implement `foldLeft[A,B]` you will be calling `foldRight[A,B=>B] (... , ...) (...)` that shall compute a new function, which then needs to be called. To my best knowledge this implementation of `foldLeft` has no practical use, but it is an interesting mind twister. It also demonstrates how to use anonymous functions to synthesize and delay computations. This technique is used for many things. We shall use it to implement lazy streams in several weeks.[10]

Exercise 11 [+]. Write a function that concatenates a list of lists into a single list. Its runtime should

[5] Exercise 3.6 [Chiusano, Bjarnason 2014]
[6] Exercise 3.9 [Chiusano, Bjarnason 2014]
[7] Exercise 3.10 [Chiusano, Bjarnason 2014]
[8] Exercise 3.11 [Chiusano, Bjarnason 2014]
[9] Exercise 3.12 [Chiusano, Bjarnason 2014]
[10] Exercise 3.13 [Chiusano, Bjarnason 2014]

be linear in the total length of all lists. Use `append`, which concatenates the two lists (described in the book).[11]

Exercise 12. Write a function filter that removes elements from a list unless they satisfy a given predicate `f`.

```
def filter[A](as: List[A])(f: A =>Boolean) : List[A]
```

Use it to remove all odd numbers from a `List[Int]` (in REPL).[12]

Exercise 13 [+]. Write a function `flatMap` that works like map except that the function given will return a list instead of a single result, and that list should be inserted into the final resulting list. Here is its signature:

```
def flatMap[A,B] (as: List[A]) (f: A =>List[B]): List[B]
```

For instance, `flatMap (List(1,2,3)) (i =>List(i,i))` should result in `List(1,1,2,2,3,3)`. Test your solution in the REPL; `flatMap` will be key in the rest of the course (together with `map`).[13]

Exercise 14. Use `flatMap` to implement `filter`. Both are standard HOFs in Scala's libraries. They were also introduced in the slides and in the book.[14]

Exercise 15 [–]. Write a function that accepts two lists and constructs a new list by adding corresponding elements. For example, `List(1,2,3)` and `List(4,5,6,7)` become `List(5,7,9)`.Trailing elements of either list are dropped.[15]

Exercise 16 [–]. Generalize the function you just wrote so that it is not specific to integers or addition. It should work with arbitrary binary operations. Name the new function `zipWith`.[16]

Exercise 17 [+]. Implement a function `hasSubsequence` for checking whether a `List` contains another `List` as a subsequence. For instance, `List(1,2,3,4)` would have `List(1,2)` , `List(2,3)`, and `List(4)` as subsequences, among others. You may have some difficulty finding a concise purely functional implementation that is also efficient. That's okay. Implement the function however comes most naturally. Note: Any two values `x` and `y` can be compared for equality in Scala using the expression `x ==y`. Here is the suggested type:

```
def hasSubsequence[A](sup: List[A], sub: List[A]): Boolean
```

Recall that an empty sequence is a subsequence of any other sequence.[17]

Exercise 18 [+]. Recall the structure of Pascal's triangle structure (this animation summarizes the key information needed: https://upload.wikimedia.org/wikipedia/commons/0/0d/PascalTriangleAnimated2.gif). Write a recursive function `pascal (n :Int) :List[Int]` that generates the nth row of Pascal's triangle. For example, `pascal(1)` should generate `List(1)`, `pascal(2)` should generate `List(1,1)`, `pascal(3)` should generate `List(1,2,1)` and `pascal(4)` should generate `List(1,3,3,1)`.

[11] Exercise 3.15 [Chiusano, Bjarnason 2014]
[12] Exercise 3.19 [Chiusano, Bjarnason 2014]
[13] Exercise 3.20 [Chiusano, Bjarnason 2014]
[14] Exercise 3.21 [Chiusano, Bjarnason 2014]
[15] Exercise 3.22 [Chiusano, Bjarnason 2014]
[16] Exercise 3.23 [Chiusano, Bjarnason 2014]
[17] Exercise 3.24 [Chiusano, Bjarnason 2014]